Dynamic ordered sets with approximate queries, approximate heaps and soft heaps

Thorup, Mikkel; Zamir, Or; Zwick, Uri

Published in:
46th International Colloquium on Automata, Languages, and Programming, ICALP 2019

DOI:
10.4230/LIPIcs.ICALP.2019.95

Publication date:
2019

Document version
Publisher's PDF, also known as Version of record

Dynamic Ordered Sets with Approximate Queries, Approximate Heaps and Soft Heaps

Mikkel Thorup
Department of Computer Science, University of Copenhagen, Denmark
mikkel2thorup@gmail.com

Or Zamir
Blavatnik School of Computer Science, Tel Aviv University, Israel
orzamir@mail.tau.ac.il

Uri Zwick
Blavatnik School of Computer Science, Tel Aviv University, Israel
zwick@tau.ac.il

Abstract

We consider word RAM data structures for maintaining ordered sets of integers whose SELECT and RANK operations are allowed to return *approximate* results, i.e., ranks, or items whose rank, differ by less than Δ from the exact answer, where $\Delta = \Delta(n)$ is an error parameter. Related to approximate SELECT and RANK is approximate (one-dimensional) NEAREST-NEIGHBOR. A special case of approximate SELECT queries are approximate MIN queries. Data structures that support approximate MIN operations are known as *approximate heaps* (priority queues). Related to approximate heaps are *soft heaps*, which are approximate heaps with a different notion of approximation.

We prove the optimality of all the data structures presented, either through matching cell-probe lower bounds, or through equivalences to well studied static problems. For approximate SELECT, RANK, and NEAREST-NEIGHBOR operations we get matching cell-probe lower bounds. We prove an equivalence between approximate MIN operations, i.e., approximate heaps, and the static *partitioning* problem. Finally, we prove an equivalence between soft heaps and the classical *sorting* problem, on a smaller number of items.

Our results have many interesting and unexpected consequences. It turns out that approximation greatly speeds up some of these operations, while others are almost unaffected. In particular, while SELECT and RANK have identical operation times, both in comparison-based and word RAM implementations, an interesting separation emerges between the approximate versions of these operations in the word RAM model. Approximate SELECT is much faster than approximate RANK. It also turns out that approximate MIN is *exponentially* faster than the more general approximate SELECT. Next, we show that implementing soft heaps is harder than implementing approximate heaps. The relation between them corresponds to the relation between sorting and partitioning.

Finally, as an interesting byproduct, we observe that a combination of known techniques yields a *deterministic* word RAM algorithm for (exactly) sorting n items in $O(n \log\log_w n)$ time, where w is the word length. Even for the easier problem of finding duplicates, the best previous deterministic bound was $O(\min\{n \log\log n, n \log_w n\})$. Our new unifying bound is an improvement when w is sufficiently large compared with n.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases Order queries, word RAM, lower bounds

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.95

Category Track A: Algorithms, Complexity and Games

Funding Mikkel Thorup's research is supported by his Advanced Grant DFF-0602-02499B from the Danish Council for Independent Research and by his Investigator Grant 16582, Basic Algorithms Research Copenhagen (BARC), from the VILLUM Foundation. Part of this research was carried out while Or Zamir and Uri Zwick were visiting BARC, Copenhagen, funded by the VILLUM Foundation, grant 16582. Part of the research was carried out while Uri Zwick was visiting IRIF, Paris, with support from the FSMP.

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 95; pp. 95:1–95:13

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

A data structure for maintaining *dynamic ordered sets* supports the operations INSERT, DELETE, and either, or both, of the operations SELECT and RANK. INSERT inserts an item, with an associated key, into the set. We assume that the keys of all items are distinct. DELETE receives a reference to an item in the set and deletes it from the set. SELECT receives an index i and returns the i-th item in the sorted order of the items currently in the set. RANK receives an item, not necessarily in the set, and returns its *rank*, i.e., the number of items in the set whose keys are smaller than the key of the item. If both SELECT and RANK operations are implemented, then it is easy to use them to support other operations such as PREDECESSOR, SUCCESSOR and NEAREST-NEIGHBOR. More specifically, PREDECESSOR(x) = SELECT(RANK(x)), SUCCESSOR(x) = SELECT(RANK$(x)+1$), and NEAREST-NEIGHBOR(x) is the closer of these two to x. (We assume here that x is not in the set.)

In the comparison-based model, each operation can be implemented in $O(\log n)$ time, and this is optimal. In this paper, however, our main focus is the word RAM model. Assuming integer keys, it models what can be implemented in a programming language such as C [18] which has been used for efficient portable code since 1978. Word operations take constant time. The word size w, measured in bits, is a unifying parameter of the model. All integers considered are assumed to fit in a word. If $|S| = n$, where S is the set of items in the data structure, then we assume that $w \geq \log n$, so that we can at least index the items in S. Integers can not only be compared. We can use all the standard arithmetic and bit-wise operations. Moreover, the random access memory of the word RAM implies that we can allocate tables or arrays of words, accessing entries in constant time using indices that may be computed from keys. These word RAM features are used in many classic algorithms, e.g., radix sort from 1929 [7] and hash tables from 1956 [9]. We note that by handling integer keys, we immediately handle floating point keys since their standard bit-representation (sign,exponent,mantissa) is such that casting them as integers gives the right ordering.

In the word RAM model, a more diverse picture emerges than in the comparison-based model. Pătraşcu and Thorup [20] obtained a data structure that supports all the above operations in $O(\log n/\log w)$ time. For SELECT and RANK this matches a lower bound of Fredman and Saks [12] which holds even if we only want to support SELECT or only want to support RANK. However, if we only want to support PREDECESSOR, SUCCESSOR and/or NEAREST-NEIGHBOR (plus INSERT and DELETE), then Andersson and Thorup [2] have shown that this can done in $O\sqrt{\log n/\log\log n}$) time per operation. This is the best possible bound in terms of n, matching a static lower bound of Beame and Fich [3].

In this paper, we consider word RAM data structures for maintaining ordered sets of integers where SELECT and RANK are allowed to return *approximate* results, i.e., ranks, or items whose rank, differ by *less* than Δ from the exact answer, where $\Delta = \Delta(n)$ is an error parameter. (When $\Delta = 1$, the data structure has to return exact results.) We use Δ-SELECT as a shorthand for Δ-approximate SELECT, and similarly for the other operations.

A Δ-MIN operation is an operation that returns one of the Δ smallest items in the ordered set. This is clearly a special case of Δ-SELECT operations. A data structure that supports INSERT, DELETE and Δ-MIN operations is known as an *approximate heap*, or Δ-heap.

An EXTRACT-Δ-MIN operation finds one of the smallest Δ items, using a Δ-MIN operation, returns it, and deletes it from the Δ-heap, using a DELETE operation. Our Δ-heap data structure satisfies the following *fairness* condition: an item cannot be one of the Δ smallest items in the data structure for 2Δ consecutive EXTRACT-Δ-MIN operations without being returned, and deleted, by one of the these EXTRACT-Δ-MIN operations.

Related to approximate heaps are *soft heaps*, introduced by Chazelle [6], which are, in a sense, approximate heaps with a different notion of approximation. Soft heaps were used by Chazelle [5] to obtain the fastest deterministic algorithm for computing minimum spanning trees in the comparison model. The implementation of soft heaps was simplified by Kaplan et al. [17]. Soft heaps were also used recently by Kaplan et al. [16] to obtain simplified optimal algorithms for some selection problems.

A *soft heap* is a heap data structure which is allowed to *increase* the keys of some of the items stored in the heap. Each item has both an *original* key, and a *current* key, which might be larger than its original key. An item whose current key is larger than its original key is said to be *corrupt*. A MIN operation returns an item with the minimum current key. (Once an item becomes corrupt, it remains corrupt, as its current key can only be increased. The user can examine the original and current keys of all items, including those of the item returned by a MIN operation.) A q-soft heap is a soft heap such that after any sequence that includes n INSERT operations, at most n/q of the items *in* the heap are corrupt. Clearly, an (n/Δ)-soft heap is also a Δ-heap. We show, however, that implementing an (n/Δ)-soft heap in the word RAM model is, in general, harder than implementing a Δ-heap.

Motivation. In addition to being natural computational problems that lead to many interesting, and perhaps unexpected, theoretical results, the problems we consider are also well motivated in practice. In many real life applications, keys are obtained using inexact measurements, or may change over time. We may be interested in 'sampling' items of given ranks, e.g., for statistical purposes, but typically when we ask for an item of rank i, an item whose rank is between $i - \Delta$ and $i + \Delta$, for a small enough Δ, will serve just as well. It is thus interesting to know whether allowing approximation can speed up such operations.

We note that the direct motivation for approximate versions of SELECT, RANK and MIN, is very different from the motivation of soft heaps which at first may look a bit peculiar, but which have proven to be useful data structure inside some important algorithms.

Our results

We show, among other things, that if $\Delta = n^\varepsilon$, for any $\varepsilon > 0$, then INSERT, DELETE and Δ-SELECT can be implemented in *constant time*.

While SELECT and RANK operations have the same running times in the word RAM model when exact results are required, an interesting separation emerges between the approximate versions of these operations. Surprisingly, Δ-SELECT is *easier* than Δ-RANK. We also show that Δ-MIN is "exponentially" faster than Δ-SELECT. As mentioned, we also show that soft heaps are harder to implement than approximate heaps.

Our results follow from a full characterization of the time needed for each subset of dynamic operations via either a matching cell-probe lower bound, or via an equivalence to the well-studied exact static problems of *ordered partition* and *sorting*.

More specifically, we obtain the following four main results:

(1) A data structure that supports INSERT, DELETE and Δ-SELECT in $O(\log n/\log(w\Delta))$ time, where n is the number of items in the set. We also obtain a matching lower bound that shows that at least one of these operations must take $\Omega(\log n/\log(w\Delta))$ time. For $\Delta = n^\varepsilon$, for any $\varepsilon > 0$, we get $O(1)$ time for each operations.

(2) An augmentation of the previous data structure that also supports Δ-RANK operations. INSERT, DELETE and Δ-SELECT operations still take $t_u = O(\log n/\log(w\Delta))$ time, while Δ-RANK takes $O\left(\log n/\log(w\Delta) + pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$ time, where $pred(n, t, s, w)$ is the

time for dynamically answering exact *predecessor queries* on a set of n items when the update time is $O(t)$, the space of the data structure is $O(s)$, and the word length is w. (It is known, for example that $pred(n,t,n,w) = O\left(\sqrt{\frac{\log n}{\log\log n}}\right)$, for every $w \geq \log n$, where $t = O\left(\sqrt{\frac{\log n}{\log\log n}}\right)$, and that this is the optimal bound in terms of n.) We also provide a matching lower bound for the complexity of Δ-RANK operations that shows that the $pred(\frac{n}{\Delta}, t_u\Delta, n, w)$ term in the upper bound cannot be removed. For $\Delta = n^\varepsilon$, for any $\varepsilon > 0$, we get $O(1)$ time for Δ-SELECT, while Δ-RANK operations still cost $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$. This exhibits a somewhat surprising separation between Δ-SELECT and Δ-RANK, given that the exact versions of these operations have the same running times.

(3) A three-way equivalence between (i) *approximate heaps*, (ii) *approximate sorting* and (iii) (exact) *ordered partition*: There is a Δ-heap with $O(P_\Delta(n,w))$ time per operation, where $P_\Delta(n,w)$ is *non-decreasing* in n, if and only if it is possible, for every n, to Δ-sort a set of n items in $O(nP_\Delta(n,w))$ time, if and only if it is possible, for every n, to partition n items into $\frac{n}{\Delta}$ sets of size roughly Δ in $O(nP_\Delta(n,w))$ time, such that the items in each set are smaller than the items in the next set. (A sequence is Δ-sorted if no item is at distance greater than Δ from its position in the sorted sequence.)

(4) A three-way equivalence between (i) *soft heaps*, (ii) *exact heaps* and (iii) exact sorting: There is q-soft heap with $O(t)$ time per operation, if and only if there is an *exact* heap holding up to q items with $O(t)$ time per operation, if and only if it is possible to sort up to q items in $O(t)$ time per item.

Han and Thorup [15] showed that n items can be ordered partitioned into $\sqrt{n}$ sets of size roughly $\sqrt{n}$ in $O(n)$ time. By iterating, if follows that n items can be partitioned into sets of size roughly $n^{2^{-k}}$ in $O(kn)$ time. Thus, n items can be partitioned into subsets of size Δ in $O(n\log(\frac{\log n}{\log\Delta}))$ time. If $\Delta < w$ we can do even better. We only do $\log\frac{\log n}{\log w}$ partitioning iterations after which we are left with sets of size roughly w. These sets can be completely sorted in linear time using the *dynamic fusion node* of Pătraşcu and Thorup [20]. Combining these two results, we get that partitioning into sets of size Δ can be done in $O(n\log(\frac{\log n}{\log(w\Delta)}))$ time. In particular, for $\Delta = 1$, we get exact sorting in $O(n\log\log_w n)$ time, a bound that has not been observed before. By (3), there is a Δ-heap with $O(\log(\frac{\log n}{\log(w\Delta)}))$ time per operation. By the lower bound in (1), Δ-SELECT requires $\Omega((\log n)/\log(w\Delta))$ time. It follows that $time(\Delta\text{-MIN}) = O(\log time(\Delta\text{-SELECT}))$, i.e., Δ-MIN is "exponentially" faster than Δ-SELECT. To get a constant time for Δ-MIN, we currently need $\Delta = n^\varepsilon$, for some $\varepsilon > 0$, as for Δ-SELECT. But, while the result for Δ-SELECT is optimal, and hence cannot be improved, improved partitioning algorithms could potentially yield $O(1)$ time of Δ-MIN for smaller values of Δ.

The bounds we give in this paper are amortized. However, all claimed time bounds can be made worst-case using the techniques of Andersson and Thorup [2]. All our data structures use linear space.

Dumitrescu [10] and Fredman [11] considered *comparison-based* data structures that support Δ-SELECT and Δ-MIN operations. The optimal time bounds for these operations are $\Theta(\log\frac{n}{\Delta})$. Δ-RANK operations can be easily supported within the same time bounds. Thus, there is no separation between Δ-SELECT and Δ-RANK in this model, and to get a constant time per operation, Δ has to be *linear* in n.

A summary of our results for Δ-SELECT, Δ-NEAREST-NEIGHBOR and Δ-RANK are given in Table 1. Update time refers to the time of INSERT and DELETE operations. The first row gives the result of a general value of Δ. The query times are optimal given the update times and assuming $O(n)$ space. (We have no proof that the same query times cannot be obtained with

Error	Update time	Δ-SELECT	Δ-NEAREST	Δ-RANK
Δ	$t_u = \Theta\left(\frac{\log n}{\log(w\Delta)}\right)$	$\Theta\left(\frac{\log n}{\log(w\Delta)}\right)$	$\Theta\left(pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$	$\Theta\left(\begin{array}{c}\Delta\text{-SELECT} + \\ \text{NEAREST}\end{array}\right)$
$\Delta = \log n$	$\Theta\left(\frac{\log n}{\log\log n}\right)$	$\Theta\left(\frac{\log n}{\log\log n}\right)$	$\Theta\left(\sqrt{\frac{\log n}{\log\log n}}\right)$	$\Theta\left(\frac{\log n}{\log\log n}\right)$
$\Delta = \sqrt{n}$	$\Theta(1)$	$\Theta(1)$	$\Theta\left(\sqrt{\frac{\log n}{\log\log n}}\right)$	$\Theta\left(\sqrt{\frac{\log n}{\log\log n}}\right)$
$\Delta = \frac{n}{w}$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Figure 1 Running times for Δ-RANK, and Δ-SELECT and Δ-NEAREST-NEIGHBOR queries.

smaller update times.) The second and third rows specialize the results for the representative cases $\Delta = \log n$ and $\Delta = \sqrt{n}$, giving bounds that hold for all values of $w \geq \log n$. With regard to the second row, we note that the time bounds obtained for $\Delta = \log n$ are identical to the exact case, i.e., $\Delta = 1$. We also note that if only Δ-NEAREST-NEIGHBOR queries are to be answered, then the same query time of $\Theta\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ can be obtained with a reduced update time of $\Theta\left(\sqrt{\frac{\log n}{\log\log n}}\right)$. (See [2].) In the third row $\Delta = \sqrt{n}$ can be replaced by $\Delta = n^\varepsilon$, for every $\varepsilon > 0$. Finally, in the forth row, we consider the case $\Delta = \frac{n}{w}$ (or equivalently $\Delta = \frac{n}{w^k}$, for any $k \geq 1$), where all update and query time drop down to a constant.

Chazelle [6], and Kaplan et al. [17], obtained $(1/\varepsilon)$-soft heaps with $O(\log\frac{1}{\varepsilon})$ amortized time per operation, which is optimal in the comparison-based model. In the word-RAM, we obtain by (4) $(1/\varepsilon)$-soft heaps with $O(\log\log\frac{1}{\varepsilon})$ amortized time per operation, or even $O(\sqrt{\log\log\frac{1}{\varepsilon}})$ amortized expected time per operation. We also obtain a w-soft heap with $O(1)$ time per operation.

The rest of the paper is organized as follows. In Section 2 we give a high-level description of the data structures and the equivalences obtained in this paper. The full details are given in the full version of the paper. In Section 3 we present our matching cell-probe lower bounds. We end in Section 4 with some concluding remarks and open problems.

2 High-level description of data structures and equivalences

In this section we sketch the techniques we use and give high level descriptions of the data structures and equivalences obtained in the paper.

2.1 Dynamic sets with approximate SELECT

Our goal is to obtain a data structure that supports INSERT, DELETE and Δ-SELECT operations in $O(\log n/\log(w\Delta))$ time per operation, which we later show is optimal. As mentioned, Pătraşcu and Thorup [20] obtained a data structure that supports INSERT, DELETE and exact SELECT operations in $O(\log n/\log w)$ time. Thus, we may assume that, say, $\Delta > w^3$ and devise a data structure that supports each operation in $O(\log n/\log\Delta)$ time.

Our data structure is built around a B-tree (see, e.g., [8]). The degree DEGREE$[v]$ of each node v in the tree is in the range $[\frac{1}{4}D, 4D]$, where $D = \Delta^{1/3}$. As with standard B-trees, a node v of degree d contains an array CHILD$[v]$ of size d with pointers to its d children,

a pointer PARENT[v] to its parent, and an array SPLIT[v] of $d-1$ *splitters*, $s_1, s_2, \ldots, s_{d-1}$. We also let $s_0 = -\infty$ and $s_d = \infty$. The keys of all items in the subtree of the i-th child CHILD[v][i] of v are all in the range $[s_i, s_{i+1})$, for $i = 0, 1, \ldots, d-1$. A non-root node v also contains its *index* INDEX[v] such that $v =$ CHILD[PARENT[v]][INDEX[v]], i.e., v is the INDEX[v]-th child of its parent. The leaves of a B-tree are all at the same depth.

The B-tree used differs from a standard B-tree in several important ways. The first is that the leaves of the tree do not contain single items, but rather *buckets* that contain between Δ and 2Δ items. These buckets are referred to as *leaf buckets*. The items in each leaf bucket are not sorted, but they all lie between the appropriate splitters in the non-leaf nodes of the tree. Second, each internal node v of the tree has a *buffer* BUFFER[v] associated with it. The size of each such buffer is at most $B = \Delta^{2/3}$. The operations on the B-tree are done *lazily*. An inserted item is simply placed at the buffer of the root. When a buffer is full, its items are partitioned according to the splitters stored in the node, and sent to the appropriate children. We refer to this operation as *flushing* the buffer.

All the items in the data structure reside in leaf buckets and buffers. The splitters in the internal nodes of the tree are copies of keys of items that belonged to the data structure at some stage.

The partitioning of the items in a buffer is done using the fast partitioning algorithm of Han and Thorup [15]. Their algorithm partitions q items according to $O(\sqrt{q})$ splitters in $O(q)$ time. The choice $B = \Delta^{2/3}$ and $D = \Delta^{1/3}$ ensures that a buffer can be flushed in $O(B)$ time. The use of this fast partitioning algorithm is the *only* place in which the data structure relies on the power of the word RAM model.

Another difference between the B-tree used and a standard one is that we impose explicit conditions on the *size* of each subtree. The size of a subtree is the total number of items in the leaf buckets and the buffers that belong to the subtree. The size of a subtree of height i (where the leaves are at height 0), is required to be in the range $[\frac{1}{2}\Delta, 2\Delta]D^i$. To maintain this condition, we store at each node v a $size[v]$ field that holds its current size. (To achieve that, $size[v]$ is updated by relevant INSERT and DELETE operations.) A simple calculation shows that the size condition implies that the degree of each node is in the range $[\frac{D}{4}, 4D]$. It also implies that the height h of the tree is at most $h \leq \log_D \frac{2n}{\Delta} \leq \frac{3 \log n}{\log \Delta}$, as $D = \Delta^{1/3}$. We thus need to show that the (amortized) cost of each INSERT, DELETE and Δ-SELECT is of the order of the depth of the tree.

To support approximate SELECT operations, we augment the B-tree by several additional components. To each non-leaf node v we add a *(rough) locator* array LOCATE[v] of size $\lfloor size_0[v]/(\frac{1}{2}\Delta D^{i-1}) \rfloor \leq 4D$, where $size_0[v]$ is the size of v when it was last rebuilt (see below). If v is at height i, where $i > 0$, then the j-entry of the array, for $j = 0, 1, \ldots, 4D-1$, contains the index of the child of v that contained the item of rank $\frac{1}{2}\Delta D^{i-1} \cdot j$ in the subtree rooted at v, i.e., the $(\frac{1}{2}\Delta D^{i-1} \cdot j)$-th item in the sorted order of all items in the subtree of v, when this subtree was last rebuilt. INSERT and DELETE operations performed after the last rebuilding of the subtree of v make the information in LOCATE[v] slightly inaccurate, but to an extent that can be tolerated, as we are only aiming for approximate results.

Finally, we also store at each non-leaf node v of degree d an array SUM[v] of size $d+1$ such that SUM[v][i], for $i = 0, 1, \ldots, d$, is the sum of sizes of the subtrees rooted at the first i children of v at the time the subtree of v was last rebuilt.

Using this augmented B-tree we can implement INSERT, DELETE and Δ-SELECT operations in $O(\log n / \log \Delta)$ amortized time. The challenge is to time the flushing of buffers and the rebuilding of subtrees so that, on the one hand, we do not spend too much time, and, on the other hand, the information in the B-tree is always sufficiently accurate so that the rank error in each SELECT operation is at most Δ.

Essentially, to locate an item whose rank is close to k, we navigate the tree using the LOCATE and SUM arrays. We start with v being the root and i being the height of the root. To find the child we need to descend to, we let $j \leftarrow \text{LOCATE}[v][\lfloor k/(\frac{1}{2}\Delta D^{i-1})\rfloor]$. It is easy to see that the k-th item in the subtree of v, at the time of the last rebuilding, is either contained in the j-th child of v, if $\text{SUM}[v][j] \leq i < \text{SUM}[v][j+1]$, or otherwise in the $(j+1)$-st child of v. (This follows as $k \leq (\frac{1}{2}\Delta D^{i-1})\lceil k/(\frac{1}{2}\Delta D^{i-1})\rceil$.) In the latter case, we increment j. We now descend to the j-th child of v, letting $v \leftarrow \text{CHILD}[v][j]$, $i \leftarrow i-1$, $k \leftarrow k - \text{SUM}[r][j]$, and repeat the process from there until we get to a leaf. We then return an arbitrary item contained in the corresponding leaf bucket. In the full version of the paper we analyze this process and show that the rank of the returned item is close enough to k.

To satisfy the *fairness* condition mentioned above, two minor changes are needed: (1) the insertion buffer should be emptied once for every Δ updates (not just every Δ insertions); (2) the base sets are maintained as queues (FIFO).

It is possible to convert the amortized time bounds into worst-case time bounds using the techniques of Andersson and Thorup [2].

2.2 Dynamic sets with approximate SELECT and RANK

Pătraşcu and Thorup [20] devised a data structure that supports INSERT, DELETE and *exact* SELECT and RANK operations in $O(\log n/\log w)$ time. An interesting separation between SELECT and RANK operations emerges when approximate results are allowed. In this section we explain how the data structure of Section 2.1 can be extended to support Δ-RANK operations. While the time of INSERT, DELETE and Δ-SELECT operations remains unchanged, i.e., $O(\log n/\log(w\Delta))$, the time of Δ-RANK operations is $O\left(\log n/\log(w\Delta) + pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$ time, where $pred(n,t,s,w)$ is the time for dynamically answering *predecessor queries* on a set of n items when the update is $O(t)$, the space used by the data structure is $O(s)$, and the word length is w. We later give a lower bound that shows that appearance of the $pred(\frac{n}{\Delta}, t_u\Delta, n, w)$ term here cannot be avoided.

Quite a lot is known about $pred(n,t,s,w)$, the time for answering exact PREDECESSOR queries. Beame and Fich [3] gave $\Omega\left(\sqrt{\frac{\log n}{\log\log n}}\right)$ and $\Omega\left(\frac{\log w}{\log\log w}\right)$ lower bounds for the static version of the problem, where polynomial time preprocessing and polynomial space are allowed. (Static means no updates.) Pătraşcu and Thorup [19] obtained a complete query-space tradeoff, when again no updates are allowed, i.e., they determined $pred(n,\infty,s,w)$ asymptotically for all s and w. Pătraşcu and Thorup [19] showed that $pred\left(n, \frac{\log n}{\log w}, n, w\right) = \Theta\left(\frac{\log n}{\log w}\right)$ and that $pred(w,1,w,w) = O(1)$. Further complications arise when deterministic vs. randomized variants of the problem are considered. We refer the reader to [19, 20] for the exact details. The picture simplifies considerably when $pred(n,t,s) = \max_w pred(n,t,s,w)$ is considered. Here, it is known that $pred(n, \sqrt{\frac{\log n}{\log\log n}}, n, w) = pred(n, n^{O(1)}, n^{O(1)}) = \Theta\left(\sqrt{\frac{\log n}{\log\log n}}\right)$.

As in Section 2.1, we may assume that $\Delta > w^3$, as the case $\Delta \leq w^3$ is covered by the exact algorithm of Pătraşcu and Thorup [20], and aim for $t_u = O(\log n/\log\Delta)$ and $O\left(\log n/\log\Delta + pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$ bounds, respectively.

It is not difficult to see that the rank of an item appearing in a leaf bucket or as a splitter in the data structure of Section 2.1 can be easily approximated in $O(\log n/\log\Delta)$ time. A Δ-RANK operation, however, must also be able to return the (approximate) rank of an item that does not appear in the data structure. To achieve that, we maintain an exact dynamic predecessor data structure on the splitters. Given an item not in the data structure we

do a PREDECESSOR query on the key of the item, and return the approximate rank of the returned splitter. As there are only $O(\frac{n}{\Delta})$ splitters, the total time of a Δ-RANK operation is $O\left(\log n/\log\Delta + pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$, as required. As we need to update the predecessor data structure, on average, only once in every Δ operations, we can afford to spend $t_u\Delta$ time on updates to the predecessor data structure. In Section 3, we show that the use of a predecessor data structure is essential.

2.3 Equivalence between approximate heaps and partitioning

The MIN operation is a very special SELECT operation in which the smallest, i.e., the item of rank 0, is sought after. Thus, any data structure that supports approximate SELECT operations also support approximate MIN operations. However, MIN operations could potentially be faster than general SELECT operations. This is true, for example, for the exact versions of these problems on the word RAM.

A data structure that supports INSERT, DELETE and MIN operations is a *priority queue*. A Δ-heap is a data structure that supports Δ-MIN operations which return one of the smallest Δ items in the data structure. (For $\Delta = 1$ we get an exact priority queue.)

Thorup [21] obtained an equivalence between priority queues and sorting. Namely, there is an (exact) priority queue that supports each operation in $P(n)$ time, where $P(n)$ is non-decreasing in n, if and only if it is possible, for every n, to sort n items in $O(n\,P(n))$ time. Here we extend this result to obtain an equivalence between Δ-heaps and the *ordered* Δ*-partition* problem: Given n items, partition them into $k \approx \frac{n}{\Delta}$ sets $A_1, A_2, \ldots, A_k$, each of size about Δ, such that $A_i < A_{i+1}$, i.e., all items in A_i are smaller than all items in A_{i+1}, for $i = 0, 1, \ldots, k-1$. We show that there is a Δ-heap with $O(P_\Delta(n))$ time per operation, where $P_\Delta(n)$ is non-decreasing in n, if and only if it is possible to Δ-partition a set of n items in $O(nP_\Delta(n))$ time.

Han and Thorup [15] showed that the ordered Δ-partition problem is equivalent to the problem of partitioning a set A of size n according to $k \approx \frac{n}{\Delta}$ splitters $s_1 < s_2 < \ldots < s_k$, producing sets $A_0, A_1, \ldots, A_k$ such that $s_i \le A_i < s_{i+1}$, for $i = 0, 1, \ldots, k$, where $s_0 = -\infty$ and $s_{k+1} = +\infty$. Note that in this variant the sets $A_0, A_1, \ldots, A_k$ are *not* necessarily of (roughly) equal size. This is the version of the ordered Δ-partition problem that we use in this section.

We sketch here the construction of a $O(\Delta \log n)$-approximate heap using an algorithm for the ordered Δ-partition problem, which is the interesting direction of the equivalence. In the full version of the paper we complete the details of this construction, remove the $O(\log n)$ from the approximation factor, and prove the other direction of the equivalence.

Most of the items in the priority queue constructed are stored in *buckets* $A_0, A_1, \ldots, A_k$ separated by $k+2$ *base splitters* $-\infty = s_0 < s_1 < s_2 < \ldots < s_k < s_{k+1} = +\infty$. Thus, $s_0 \le A_0 < s_1 \le A_1 < \ldots < s_k \le A_k < s_{k+1}$. The splitters are copies of keys of items that belong or belonged to the priority queue at some stage. The items within each bucket are unsorted. We let $\Phi = \Delta \log n$ and require that $\frac{\Phi}{4} \le |A_i| \le \Phi$, for $i = 0, 1, \ldots, k-1$, and $|A_k| \le \Phi$.

A logarithmic number of splitters $t_0 < t_1 < \ldots < t_{\ell+1}$, where $\ell = \Theta(\log n)$, also serve as *level splitters*. We have $t_0 = s_0 = -\infty$ and $t_{\ell+1} = s_{k+1} = +\infty$. Each level splitter t_j has a *level buffer* B_j associated with it that can hold up to $4^j\Delta$ items. We also maintain a counter q_j that provides an upper bound of the size of B_j. Finally, there is also an *insertion buffer* $B = B_0$ that can hold up to Φ items. We maintain the following invariants, for $j = 1, 2, \ldots \ell$:

(i) There are more than $\frac{1}{2}4^j \cdot \Phi$ bucket items smaller than t_j, if $t_j < \infty$.
(ii) There are less than $\frac{7}{4}4^j \cdot \Phi$ items smaller than t_j.
(iii) The keys of all items in B_j are in $[t_j, t_{j+2})$. Here $t_{\ell+1} = t_{\ell+2} = \infty$.

To insert an item, we simply add it to the insertion buffer. If the insertion buffer is not full, this completes the operation. To delete an item, given a pointer to it, we simply delete it from its bucket or buffer, and update some counters. A MIN operation returns an arbitrary item from the bucket A_0, referred to as the *head*. The rank of the returned item is at most $2\Phi = 2\Delta \log n$, as only items in the head and the insertion buffer may be smaller than the returned item.

When the insertion buffer B is full, i.e., it contains $\Phi = \Delta \log n$ items, we use the ordered Δ-partition algorithm to split the items in B according to the $O(\log n)$ level splitters. The amortized cost per item is $O(P_\Delta(\Delta \log n)) = O(P_\Delta(n))$. Recall that $O(P_\Delta(n))$ is the time, per item, needed to order partition n items into sets of size roughly Δ, or equivalently, to Δ-sort n items. (We may assume that $\Delta \leq n^{1/2}$, as $\Delta = n^{1/2}$ already yields a constant time per operation, and hence $\Delta \log n \leq n$.) Let $B^1 < B^2 < \ldots < B^\ell$ be the resulting partition. The items in B^j are added, one by one, into the level buffer B_j.

When a level buffer B_j is full, or when we need to change the level splitter t_j, we split its items according to all base splitters that are smaller than t_{j+2}. As the number of items smaller than t_{j+2} is $O(4^j\Phi)$, and as each bucket is of size $\Theta(\Phi)$, the number of splitters smaller than t_{j+2} is $O(4^j)$. The number of items in the buffer is at most $4^j\Delta$. Thus, the splitting can be done in $O(4^j\Delta \cdot P_\Delta(n))$ time. The items in each set of the partition generated are added, one by one, to the appropriate buckets.

The above "idyllic" description ignored the fact that base buckets may get too large or too small, and that the three invariants imposed on the level splitters may be violated. However, as we allowed enough slack in the size of the base buckets and in the invariants, it is not too difficult to fix these problems by merging and splitting adjacent base buckets, and by periodically redefining the level splitters. The fairly technical details are given in the full version of the paper. The amortized time per operation remains $O(P_\Delta(n))$.

2.4 Equivalence between soft heaps and sorting

A q-soft heap is a heap which is allowed to corrupt, i.e., increase the keys, of a small fraction of its items. More specifically, after n INSERT operations, at most n/q of the items *in* the heap are allowed to be corrupt. The hope, of course, is that allowing corruptions leads to a more efficient implementation of the heap operations. (It is important to note that the number of corrupt items is related to the number of insertions, not to the number of items currently in the heap.)

It follows easily from the definition that as long as less than q items are inserted into a q-soft heap, the heap is not allowed to corrupt any item. Thus, a q-soft heap with $O(t)$ time per operation can be used to exactly sort $q-1$ items in $O(qt)$ time. By Thorup's [21] equivalence between sorting and priority queues, we get that if there is q-soft heap with $O(t)$ time per operation, then there is also an *exact* heap that can hold up to q items with $O(t)$ time per operation. It is also easy, to obtain this result directly. To implement an exact heap on at most q items, we use a $2q$-soft heap. After every $2q$ insertions we rebuild the $2q$-soft heap using at most q insertions. The amortized time per operation stays $O(t)$.

More surprising is that we also have the opposite implication. If there is an exact heap on at most q items with $O(t)$ time per operation, then there is also a q-soft heap with $O(t)$ time per operation. (Note that the number of items in the q-soft heap is not bounded.) To

prove this result we use the simplified construction of soft heaps by Kaplan et al. [17]. We observe that certain components in this comparison-based implementation of soft heaps can be replaced by exact heaps that hold up to q items. This also leads to a clearer understanding of how soft heaps work.

2.5 Faster exact sorting

Han and Thorup [15] obtained a deterministic linear time algorithm for partitioning n items into a sequence of $\sqrt{n}$ sets, each of size roughly $\sqrt{n}$, such that the items in each set are smaller than the items in the next set. This leads, by repeated partitioning, to a deterministic $O(n \log\log n)$-time algorithm. A different deterministic $O(n \log\log n)$-time sorting algorithm was earlier obtained by Han [14].

More recently, Pătraşcu and Thorup [20], improving results of Fredman and Willard [13], implemented *dynamic fusion trees*, supporting INSERT, DELETE, RANK and SELECT in $O(\log_w n)$ time per operation, where n is the size of the set and w is the word length. This leads immediately to an $O(n \log_w n)$-time sorting algorithm. This algorithm is faster than the $O(n \log\log n)$-time algorithm for sufficiently large w, e.g., $w = n^{\omega(1/\log\log n)}$.

A simple combination of these two algorithms gives rise to a deterministic $O(n \log\log_w n)$-time algorithm. This bound subsumes the two previous bounds. Perform $\log\log_w n$ partitioning steps of [15] that partition the n input items into sets of size w. These sets are then sorted in linear time using the dynamic fusion trees of [20]. The $O(n \log\log_w n)$-time algorithm is asymptotically faster than the $O(n \log\log n)$-time algorithm for much smaller values of w, namely $w = n^{1/\log^{o(1)} n}$.

Han and Thorup [15] obtain a faster randomized $O(n\sqrt{\log\log n})$-time sorting algorithm. Andersson et al. [1] and Belazzougui et al. [4] have shown that sorting can be done in linear expected time when $w = \Omega(\log^2 n \log\log n)$.

It is an open problem whether randomization can speed up partitioning algorithms and lead, in particular, to faster data structures for the various operations considered in this paper, such as Δ-SELECT.

3 Lower bounds

In this section we give cell-probe lower bounds that match the upper bounds obtained by the data structures for approximate SELECT and RANK given in Sections 2.1 and 2.2.

3.1 Lower bound for approximate SELECT

Pătraşcu and Thorup [20], relying on previous results of Fredman and Sacks [12], proved the following cell-probe lower bound.

▶ **Theorem 3.1.** *For any cell-probe data structure that supports* INSERT, DELETE *and (exact)* SELECT *(or* RANK*) operations, if both* INSERT *and* DELETE *require $t = t(n)$ time per operation, then* SELECT *(or* RANK*) operations must take $\Omega\left(\frac{\log n}{\log(w \cdot t(n))}\right)$ time.*

Relying on this result, using a simple reduction, we obtain our lower bound for Δ-SELECT.

▶ **Theorem 3.2.** *For any cell-probe data structure that supports* INSERT, DELETE *and* Δ-SELECT *(or* Δ-RANK*) operations, if both* INSERT *and* DELETE *require at most $O(\log n)$ time per operation, then* SELECT *(or* RANK*) operations must take $\Omega\left(\frac{\log n}{\log(w\Delta)}\right)$ time.*

Proof. Given a Δ-approximate data structure, we can construct an exact data structure by *duplicating* each item 2Δ times. More specifically, when an item is to be inserted into the exact data structure, we insert 2Δ copies of the item into the Δ-approximate data structure. When an item is to be deleted from the exact data structure, we delete its 2Δ copies from the Δ-approximate data structure. To select the item of rank i, we Δ-approximately select an item of rank $2\Delta i + \Delta$. As the rank of the item returned differs from $2\Delta i + \Delta$ by *less* than Δ, the item returned must be a copy of the item of rank i. To compute the rank of an item, not necessarily in the data structure, we do a Δ-RANK query on the approximate data structure, divide the returned rank by 2Δ and round down. It is again easy to see that this is the exact rank of the queried item.

If the times of INSERT and DELETE of the Δ-approximate data structures are $t(n)$, then the times of INSERT and DELETE in the exact data structure are $\Delta t(\Delta n)$. If $s(n)$ is the time of Δ-SELECT (or RANK), then the time for exact SELECT (or RANK) is $s(\Delta n)$. If follows from Theorem 3.1 that $s(\Delta n) = \Omega\left(\frac{\log n}{\log(w \cdot \Delta t(\Delta n))}\right)$, or equivalently $s(n) = \Omega\left(\frac{\log \frac{n}{\Delta}}{\log(w \cdot \Delta \cdot t(n))}\right)$. If $t(n) = O(\log n)$, then as $w \geq \log n$, we also get that $t(n) = O(w)$. We may also assume that $\Delta < n^{1/2}$, as otherwise, the lower bound is a constant. Thus, $s(n) = \Omega\left(\frac{\log n}{\log(w \cdot \Delta)}\right)$, as claimed. ◀

The lower bound for Δ-SELECT is tight, as it matches the upper bound of the data structure given in Section 2.2.

3.2 Lower bound for approximate RANK

The lower bound of Theorem 3.2 holds also for Δ-RANK, but it is not tight for all values of Δ. We provide here a tight lower bound that matches the performance of the data structure given in Section 2.2. The lower bound relies on the fact an exact PREDECESSOR operation can be performed using one SELECT and one RANK operations, namely, PREDECESSOR(k) = SELECT(RANK(k)).

▶ **Theorem 3.3.** *For any linear space cell-probe data structure that supports* INSERT, DELETE *and* Δ-RANK *operations, if* INSERT *and* DELETE *take at most* $t_u = O\left(\log n / \log(w\Delta)\right)$ *time per operation, then* Δ-RANK *operations must take* $\Omega\left(\log n / \log(w\Delta) + pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$ *time.*

Proof. Suppose that we are given a linear space data structure that supports INSERT and DELETE operations in $t_u = O((\log n / \log(w\Delta))$ time, and Δ-RANK operations in $r(n)$ time. We already known, by Theorem 3.2, that $r(n) = \Omega\left(\log n / \log(w\Delta)\right)$ time. Thus, we only need to show that $r(n) = \Omega\left(pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$. We show that we can use the approximate data structure given to obtain a data structure that supports exact PREDECESSOR operations in $O(r(\Delta n))$. It would then follow that $r(n) = \Omega(pred(\frac{n}{\Delta}, t_u\Delta, n, w))$.

We first add to the data structure given to us the ability to support Δ-SELECT operations. This is easily done by using *both* the given data structure *and* our Δ-SELECT data structure of Section 2.1 to hold the same set of items. INSERT and DELETE operations still take $t_u = O((\log n / \log(w\Delta))$ time, and Δ-SELECT and RANK operations still take $r(n)$ time, as we already know that $r(n) = \Omega\left(\log n / \log(w\Delta)\right)$.

We now use the duplication technique used in the proof of Theorem 3.2. We get an exact data structure in which INSERT and DELETE take $\Delta t(\Delta n)$ time, and exact SELECT and RANK, and hence exact PREDECESSOR, take $r(\Delta n)$ time. Thus, $r(n) = \Omega\left(pred(\frac{n}{\Delta}, t_u\Delta, n, w)\right)$, as claimed. ◀

4 Concluding remarks and open problems

Data structures for supporting dynamic ordered sets, i.e., data structures that support INSERT, DELETE, and either one or both of SELECT and RANK, and possibly some other operations, are among the most basic and natural data structures. Pătraşcu and Thorup [20] obtained essentially optimal implementation of such data structures in the word RAM model, the model that most closely represents what can be done on a real computer.

Surprisingly, not much attention was paid before to data structures that support *approximate* versions of these operations. There are many conceivable applications in which, for example, we do not insist on knowing the exact rank of an item in the set, a good enough approximation might be enough.

We show that allowing approximation greatly speeds up some of these operations, while the complexity of the other operations remains essentially unchanged. We obtain a full characterization of the time needed to implement approximate versions of these operations. A fairly interesting picture emerges. While the exact versions of SELECT and RANK have the same complexity, a separation emerges between the approximate versions of these problems.

We also considered approximate MIN operations that correspond to the implementation of approximate heaps (priority queues). It is known that exact MIN operations are easier than the more general SELECT operations. We show that the "exponential" gap between these operations persists when approximation is allowed. We obtained a equivalence between approximate heaps and approximate sorting, extending the equivalence of Thorup [21] for the exact versions of these problems.

Closely related to approximate heaps are Chazelle's [6] soft heaps that feature prominently in his deterministic minimum spanning tree algorithm [5]. The exact relation between approximate and soft heaps was not understood before. Looking at these two data structures using the "word RAM lens" reveals an essential difference between these two data structures. Approximate heaps correspond to approximate sorting, or partitioning, while q-soft heaps actually correspond to the *exact* sorting of sets of size q. This might explain the additional usefulness of soft heaps.

The closer look at ordered set data structures, partitioning algorithms and sorting algorithms also revealed, as a byproduct, a new deterministic sorting algorithm that runs in $O(n \log \log_w n)$ time. The new bound subsumes and improves on the two previously known bounds of $O(n \log \log n)$ and $O(n \log_w n)$.

We focused in this paper on deterministic algorithm. Studying the effect of randomization on the various problems considered is an interesting research topic. In particular, it would be interesting to know whether randomization can speed up (ordered) partitioning algorithms.

References

1 Arne Andersson, Torben Hagerup, Stefan Nilsson, and Rajeev Raman. Sorting in Linear Time? *J. Comput. Syst. Sci.*, 57(1):74–93, 1998. Announced at STOC'95. doi:10.1006/jcss.1998.1580.

2 Arne Andersson and Mikkel Thorup. Dynamic Ordered Sets with Exponential Search Trees. *J. ACM*, 54(3):Article 13, 2007. Combines results announed at FOCS'96, STOC'00, and SODA'01.

3 Paul Beame and Faith Fich. Optimal Bounds for the Predecessor Problem and Related Problems. *J. Comput. System Sci.*, 65(1):38–72, 2002. Announced at STOC'99.

4 Djamal Belazzougui, Gerth Stølting Brodal, and Jesper Sindahl Nielsen. Expected Linear Time Sorting for Word Size $\Omega(\log^2 n \log \log n)$. In *Proc. 14th SWAT*, pages 26–37, 2014. doi:10.1007/978-3-319-08404-6_3.

5 Bernard Chazelle. A minimum spanning tree algorithm with Inverse-Ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000. doi:10.1145/355541.355562.

6 Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, 2000. doi:10.1145/355541.355554.

7 L. J. Comrie. The Hollerith and Powers Tabulating Machines. *Trans. Office Machinary Users' Assoc., Ltd*, pages 25–37, 1929-30.

8 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to algorithms*. The MIT Press, 3rd edition, 2009.

9 A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

10 Adrian Dumitrescu. A Selectable Sloppy Heap. *CoRR*, abs/1607.07673, 2016. arXiv:1607.07673.

11 Michael L. Fredman. Comments on Dumitrescu's "A Selectable Sloppy Heap". *CoRR*, abs/1610.02953, 2016. arXiv:1610.02953.

12 Michael L. Fredman and Michael E. Saks. The Cell Probe Complexity of Dynamic Data Structures. In *Proc.* 21^{st} *STOC*, pages 345–354, 1989.

13 Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1993. Announced at STOC'90.

14 Yijie Han. Deterministic Sorting in $O(n \log\log n)$ Time and Linear Space. *J. Algorithms*, 50(1):95–105, 2004. Announced at STOC'02.

15 Yijie Han and Mikkel Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space. In *Proc.* 43^{rd} *FOCS*, pages 135–144, 2002.

16 Haim Kaplan, László Kozma, Or Zamir, and Uri Zwick. Selection from Heaps, Row-Sorted Matrices, and X+Y Using Soft Heaps. In *2nd Symposium on Simplicity in Algorithms, SOSA@SODA 2019, January 8-9, 2019 - San Diego, CA, USA*, pages 5:1–5:21, 2019. doi:10.4230/OASIcs.SOSA.2019.5.

17 Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. Soft Heaps Simplified. *SIAM J. Comput.*, 42(4):1660–1673, 2013. doi:10.1137/120880185.

18 B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

19 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proc.* 38^{th} *STOC*, pages 232–240, 2006. doi:10.1145/1132516.1132551.

20 Mihai Pătraşcu and Mikkel Thorup. Dynamic Integer Sets with Optimal Rank, Select, and Predecessor Search. In *Proc.* 55^{th} *FOCS*, pages 166–175, 2014. doi:10.1109/FOCS.2014.26.

21 Mikkel Thorup. Equivalence between priority queues and sorting. *J. ACM*, 54(6):28, 2007. Announced at FOCS'02. doi:10.1145/1314690.1314692.